Solutions, Exam 1, CSCI 210, Spring 2004

1. The "`grep open f > wc`" command would redirect what *grep* would normally print on the screen into a file called *wc*, whereas "`grep open f | wc`" would instead pipe this same information as input to the command *wc*, and the output of *wc* would appear on the screen. That is, the "`>`" creates a file, whereas "`|`" sends information into another command.

2. 0 000 100

3.
```
.section .data
fmt:      .string "%d"
.section .text
lastOf:   pushl %ebp                  # entry template
          movl %esp, %ebp
          subl $8, %esp               # allocate space for old ebp, k
          movl %ebx, -4(%ebp)
          movl 8(%ebp), %ebx          # use ebx to hold n

loop:     testl %ebx, %ebx            # if(n == 0) goto done;
          jz done
          leal -8(%ebp), %eax         # scanf("%d", %k);
          pushl %eax
          pushl $fmt
          call scanf
          decl %ebx                   # n--;
          jmp loop

done:     movl -4(%ebp), %esp         # restore ebx
          movl -8(%ebp), %eax         # set eax to k
          movl %ebp, %esp
          popl %ebp
          ret
```

4. Even though it uses more instructions, (b.) has fewer instructions per iteration of the loop. (All the instructions are the same, except that there is only one jump instruction (`jl`) instead of two (`jge` and `jmp`), and the `cmpl` instruction moves to the loop's bottom.) Having fewer instructions per iteration of the loop means that the total number of instructions executed will be less, even if there are actually more overall instructions in the actual code.

5. Strength reduction applies when the code contains a loop, with a counter increasing by a fixed amount each time through the loop, where the loop's body includes multiplying the counter by a value. In this case, the compiler can introduce a new variable, and add the multiplication factor to the variable each time the the counter increments.

```
i = 0;
total = 0;
while(i < n) {
    total += 5 * i;
    i++;
}
```

```
i = 0;
i5 = 0;
total = 0;
while(i < n) {
    total += i5;
    i++;
    i5 += 5;
}
```

6. Though they may in fact work identically, *gcc* will not be able to determine this, because it compiles files (and even separate functions within a file) separately. Thus, it will not look into the contents of `findGoldbach()` while it compiles `main()`, and so it must assume that `findGoldbach()` could have some effect other than simply computing a value. (It may print something to the screen, for example.) If so, then this effect would occur three times in (a.), but it would occur only once in (b.), and so the two would not be equivalent.

(Incidentally, a good programmer, who understands that they are in fact equivalent, would write (b.) rather than (a.) precisely because compilers won't be able to perform the transformation themselves.)